The Pitfalls of Benchmarking with Applications

Erven Rohou, Thierry Lafage

The Pitfalls of Benchmarking with Applications

Erven Rohou *Member, HiPEAC,* and Thierry Lafage
INRIA, Centre Inria Rennes – Bretagne Atlantique, Campus de Beaulieu, Rennes, France

***Abstract*— Application benchmarking is a widely trusted method of performance evaluation. Compiler developers rely on them to assess the correctness and performance of their optimizations; computer vendors use them to compare their respective machines; processor architects run them to tune innovative features, and — to a lesser extent — to validate their correctness. Benchmarks must reflect actual workloads of interest, and return a synthetic measure of "performance". Often, benchmarks are simply a collection of real-world applications run as black boxes. We identify a number of pitfalls that derive from using applications as benchmarks, and we illustrate them with a popular, freely available, benchmark suite. In particular, we advocate the fact that correctness should be defined by an expert of the application domain, and the test should be integrated in the benchmark.**

***Index Terms*— Benchmark, validation, correctness, performance.**

I. Introduction

A large part of the computing industry depends on benchmarks. Computer vendors, researchers in architectures, and compiler developers routinely use them. In this paper, we are mostly interested in the latter category. Compiler developers need to make sure that their optimizations produce correct code and deliver performance. However, our discussion generally applies to many categories of benchmark users.

Compiler developers rely on two categories of tests.

1) Unit tests validate a single precise functionality of the software. They are written in synergy with the application they are designed to test.
2) Benchmarks refer to a set of applications whose performance are measured in order to assess the global performance of the compiler (or of the application under development). The term *benchmark*, however, has a rather loose meaning.

The Oxford Dictionary defines a benchmark as

> "a standard or point of reference."

SPEC is more precise and defines it as

> "For computers: a benchmark is a test, or set of tests, designed to compare the performance of one computer system against the performance of others."

In the embedded systems industry, benchmarks are composed of applications developed by the company, or prototypes thereof. This is quite understandable, since the focus must be on the products sold by the company. Vendors of compilers for general purpose systems need a much broader spectrum of tests that attempt to represent all the possible customers.

In most cases, benchmarks are neither designed nor developed by the compiler development team. Some may come from the product divisions, others are bought. Some are freely available. They may also have been collected over the years by former members of the team. The key point is that they are real applications. This increases the confidence of the users who base decisions on their outcome. However, being real applications, they are complex. Their behavior is beyond the understanding of the testers who simply run them as black boxes.

As such, benchmarks are run, and their output files are compared with a reference output. What does it mean to a compiler developer if output files are not 100% correct? What about processor architects? Can a computer vendor publish performance numbers if the benchmark used to measure the speed did not produce *exactly* the same results?

In this paper, we analyze our experience with the MiBench [8] benchmark suite. Several reasons motivate our choice: the suite is freely available from the EECS Department of the University of Michigan, and it has been extended with many data sets for each benchmark (see next section). It is also widely used among the architecture and compilation research community[1], including for publication at conferences such as PLDI, CGO, HiPEAC, MICRO, ASPLOS and ISCA. As mentioned on the *download* page of the MiBench web site, some outputs are known to be specific to the architecture, "especially for benchmarks that generate floating point numbers". We investigate the variability of the output files, and analyze the reasons for it. We show that floating point computations are only one aspect of the observed variations. There are also variations on a single architecture, depending on the choice of compiler or compiler options. We also show that using poorly understood applications as benchmarks is misleading.

This paper reviews a (non-exhaustive) list of problems that derive from using real applications as benchmarks. We claim that correctness should be built in the benchmark itself, and defined by an expert of the field. Section II describes our experimental setup. We analyze our results in Section III. Related works are reviewed in Section IV and we conclude in Section V.

II. Experimental Setup

We experimented with the MiBench benchmarks [8], and in particular with the MiDataSets [6] that provide 20 different data sets for each benchmark. Table I briefly describes the benchmarks, the type of input they take, and the type of output they produce.

We considered the following target platforms:

1) x86 32-bit Linux, with GCC 4.4.1;
2) x86 64-bit Linux, with GCC 4.4.0;
3) x86 32-bit Windows 7 + cygwin, with GCC;
4) x86 32-bit Linux, with Intel icc 11.0;

[1] At the time of writing, Google Scholar reports more than 1100 citations of the original article [8]

Name	Description	Input type	Output type
bitcount	Count set bits in integer	ASCII text	ASCII text
qsort	Sorting algorithm	ASCII text	ASCII text
susan_c	Corner recognition	PGM image	PGM image
susan_e	Edge recognition	PGM image	PGM image
susan_s	Image smoothing (noise reduction)	PGM image	PGM image
jpeg_c	JPEG encoder	PPM image	JPEG image
jpeg_d	JPEG decoder	JPEG image	PPM image
lame	MP3 encoder	WAVE audio	MP3 audio
mad	MPEG audio decoding	MP3 audio	WAVE audio
tiff2bw	Conversion to black and white	TIFF image	TIFF image
tiff2rgba	Conversion to color RGB TIFF format	TIFF image	TIFF image
tiffdither	Dither a black and white picture	TIFF image	TIFF image
tiffmedian	Conversion to a reduced color palette	TIFF image	TIFF image
dijkstra	Shortest path in a graph	ASCII text	ASCII text
patricia	Create and search a Patricia trie data structure	ASCII text	ASCII text
ghostscript	PostScript interpreter	PostScript file	PPM image
ispell	Spell checker	Text, not necess. ASCII	ASCII text with `CR+LF`
rsynth	Text to speech synthesis	Text, not necess. ASCII	.AU Sun/NeXT audio
stringsearch	Search for words in text	Text, not necess. ASCII	Text, not necess. ASCII
blowfish_d	Blowfish decryption	Binary data	Binary data
blowfish_e	Blowfish encryption	Text, not necess. ASCII	Binary data
pgp_d	Asymmetric (public key) decryption	Binary data	Text, not necess. ASCII
pgp_e	Asymmetric (public key) encryption	Text, not necess. ASCII	Binary data
rijndael_d	AES decryption	Binary data	Text, not necess. ASCII
rijndael_e	AES encryption	Text, not necess. ASCII	Binary data
sha	160-bit secure hash algorithm	Text, not necess. ASCII	ASCII text
adpcm_c	Adaptive Differential Pulse Code Modulation encoder	WAVE audio	Binary data
adpcm_d	Adaptive Differential Pulse Code Modulation decoder	Binary data	Binary data
CRC32	32-bit Cyclic Redundancy Check	WAVE audio	ASCII text
gsm	GSM (Global Standard for Mobile) encode	.AU Sun/NeXT audio	Binary data

TABLE I
MIBENCH BENCHMARKS

5) SPARC 32-bit Linux, with GCC 4.3.2;
6) PowerPC 32-bit Linux, with GCC 4.4.1.

We tested optimization levels -O0, -O1, -O2, -O3 and -Os.

Our *absolute reference* is a run on the x86 32-bit Linux target compiled with GCC at -O0 optimization level. This is by no means a value judgment. One configuration has to be chosen for comparison purposes. The lowest optimization level decreases the likelihood of compiler-introduced errors. The choice of the platform is purely pragmatic: x86 Linux machines with GCC happen to be the most readily available setup in our office. In fact, as described in the next section, it would have been more appropriate to choose another platform for some benchmarks.

All benchmarks are run with this absolute reference configuration for all data sets, and the output files are stored. Successive runs with different configurations on different targets compare their outputs with this reference[2]. When the output matches the reference, the run is declared *positive*, otherwise *negative*. We later distinguish between true and false positives. A false positive is a run that produces a correct output, but should have been declared as incorrect (the output is correct by chance). A true positive refers to a correct run that produces a correct output. Similarly, a true negative is an incorrect run that produces an incorrect output (as should be), and a false negative produces an incorrect output, but should be considered correct anyway.

Tables II to VII summarize the results for each target. Each column corresponds to a configuration. A $\surd$ sign means that all 20 data sets produce the same output as the absolute reference (all runs are positive). When fewer data sets produce the expected output, the percentage of negatives is reported. Note that in Table III, some programs needed to be compiled into 32-bit code: they are flagged with `-m32`. Note that Table II has no column for -O0 since this is the reference configuration, all runs are true positives by definition.

On x86 targets, a large majority of runs are positive (i.e. produce results which compare equal to the absolute reference). Some benchmarks, however, produce 100% negative runs. The situation is worse on SPARC and PowerPC targets. The main distinguishing architectural feature is the endianness: x86 is little-endian, SPARC and PowerPC are big-endian. When output files produced on all big-endian machines are equal to each other, endianness issues can be suspected.

Two benchmarks, namely *ghostscript* and *pgp_e* crashed on SPARC and PowerPC. We could have reported this fact as 100% negative. In this particular case, the tables show "N/A".

The next section gives a more in-depth analysis of the differences in the output produced.

III. ANALYSIS

Compiler developers face the risk of introducing errors into their benchmarks. The first concern is correctness of the generated code. Performance comes second. This section adopts the same priorities: we first analyze the reasons for the failures of the benchmarks. We then discuss performance issues that may adversely impact the conclusions the benchmark users.

[2]We use the UNIX `diff` command to compare the output files.

	x86 32-bit Linux (GCC)			
	-O1	-O2	-O3	-Os
bitcount	√	√	√	√
qsort	√	√	√	√
susan_c	40%	40%	40%	40%
susan_e	√	√	√	√
susan_s	√	√	√	√
jpeg_c	√	√	√	√
jpeg_d	√	√	√	√
lame	85%	85%	85%	√
mad	√	√	√	√
tiff2bw	√	√	√	√
tiff2rgba	√	√	√	√
tiffdither	√	√	√	√
tiffmedian	√	√	√	√
dijkstra	√	√	√	√
patricia	√	√	√	√
ghostscript	√	√	√	√
ispell	√	√	√	√
rsynth	100%	100%	100%	100%
stringsearch	√	√	√	√
blowfish_d	√	√	√	√
blowfish_e	√	√	√	√
pgp_d	√	√	√	√
pgp_e	√	√	√	√
rijndael_d	√	√	√	√
rijndael_e	√	√	√	√
sha	√	√	√	√
adpcm_c	√	√	√	√
adpcm_d	√	√	√	√
CRC32	√	√	√	√
gsm	√	√	√	√

TABLE II

NEGATIVE RUNS ON X86 32-BIT LINUX (GCC)

	x86 64-bit Linux (GCC)				
	-O0	-O1	-O2	-O3	-Os
bitcount	√	√	√	√	√
qsort	√	√	√	√	√
susan_c	40%	40%	40%	40%	40%
susan_e	√	√	√	√	√
susan_s	√	√	√	√	√
jpeg_c	√	√	√	√	√
jpeg_d	√	√	√	√	√
lame	√	√	√	√	√
mad	√	√	√	√	√
tiff2bw	√	√	√	√	√
tiff2rgba	√	√	√	√	√
tiffdither	√	√	√	√	√
tiffmedian	√	√	√	√	√
dijkstra	√	√	√	√	√
patricia	√	√	√	√	√
ghostscript	5%	5%	5%	5%	5%
ispell (-m32)	√	√	√	√	√
rsynth	100%	100%	100%	100%	100%
stringsearch	√	√	√	√	√
blowfish_d	√	√	√	√	√
blowfish_e	√	√	√	√	√
pgp_d (-m32)	√	√	√	√	√
pgp_e (-m32)	√	√	√	√	√
rijndael_d	100%	100%	100%	100%	100%
rijndael_e	√	√	√	√	√
sha	100%	100%	100%	100%	100%
adpcm_c	√	√	√	√	√
adpcm_d	√	√	√	√	√
CRC32	100%	100%	100%	100%	100%
gsm	√	√	√	√	√

TABLE III

NEGATIVE RUNS ON X86 64-BIT LINUX (GCC)

	x86 32-bit Windows7+cygwin (GCC)				
	-O0	-O1	-O2	-O3	-Os
bitcount	√	√	√	√	√
qsort	100%	100%	100%	100%	100%
susan_c	√	40%	40%	40%	40%
susan_e	√	√	√	√	√
susan_s	√	√	√	√	√
jpeg_c	√	√	√	√	√
jpeg_d	√	√	√	√	√
lame	√	√	√	√	√
mad	√	√	√	√	√
tiff2bw	√	√	√	√	√
tiff2rgba	√	√	√	√	√
tiffdither	√	√	√	√	√
tiffmedian	√	√	√	√	√
dijkstra	√	√	√	√	√
patricia	√	√	√	√	√
ghostscript	√	√	√	√	√
ispell	100%	100%	100%	100%	100%
rsynth	90%	100%	100%	100%	100%
stringsearch	85%	85%	85%	85%	85%
blowfish_d	√	√	√	√	√
blowfish_e	√	√	√	√	√
pgp_d	√	√	√	√	√
pgp_e	√	√	√	√	√
rijndael_d	√	√	√	√	√
rijndael_e	100%	100%	100%	100%	100%
sha	√	√	√	√	√
adpcm_c	√	√	√	√	√
adpcm_d	√	√	√	√	√
CRC32	√	√	√	√	√
gsm	√	√	√	√	√

TABLE IV

NEGATIVE RUNS ON X86 32-BIT WINDOWS7+CYGWIN (GCC)

	x86 32-bit Linux (icc)				
	-O0	-O1	-O2	-O3	-Os
bitcount	√	√	√	√	√
qsort	√	√	√	√	√
susan_c	√	40%	40%	40%	40%
susan_e	√	√	√	√	√
susan_s	√	√	√	√	√
jpeg_c	√	√	√	√	√
jpeg_d	√	√	√	√	√
lame	√	√	√	√	√
mad	√	√	√	√	√
tiff2bw	√	√	√	√	√
tiff2rgba	√	√	√	√	√
tiffdither	√	√	√	√	√
tiffmedian	√	√	√	√	√
dijkstra	√	√	√	√	√
patricia	√	√	√	√	√
ghostscript	√	√	√	√	√
ispell	√	√	√	√	√
rsynth	100%	100%	100%	100%	100%
stringsearch	√	√	√	√	√
blowfish_d	√	√	√	√	√
blowfish_e	√	√	√	√	√
pgp_d	√	√	√	√	√
pgp_e	√	√	√	√	√
rijndael_d	√	√	√	√	√
rijndael_e	√	√	√	√	√
sha	√	√	√	√	√
adpcm_c	√	√	√	√	√
adpcm_d	√	√	√	√	√
CRC32	√	√	√	√	√
gsm	√	√	√	√	√

TABLE V

NEGATIVE RUNS ON X86 32-BIT LINUX (ICC)

	SPARC 32-bit Linux (GCC)				
	-O0	-O1	-O2	-O3	-Os
bitcount	√	√	√	√	√
qsort	√	√	√	√	√
susan_c	40%	40%	40%	40%	40%
susan_e	√	√	√	√	√
susan_s	√	√	√	√	√
jpeg_c	√	√	√	√	√
jpeg_d	√	√	√	√	√
lame	√	√	√	√	√
mad	√	√	√	√	√
tiff2bw	100%	100%	100%	100%	100%
tiff2rgba	100%	100%	100%	100%	100%
tiffdither	100%	100%	100%	100%	100%
tiffmedian	100%	100%	100%	100%	100%
dijkstra	√	√	√	√	√
patricia	√	√	√	√	√
ghostscript	N/A	N/A	N/A	N/A	N/A
ispell	100%	100%	100%	100%	100%
rsynth	100%	100%	100%	100%	100%
stringsearch	√	√	√	√	√
blowfish_d	√	√	√	√	√
blowfish_e	√	√	√	√	√
pgp_d	√	√	√	√	√
pgp_e	N/A	N/A	N/A	N/A	N/A
rijndael_d	100%	100%	100%	100%	100%
rijndael_e	100%	100%	100%	100%	100%
sha	100%	100%	100%	100%	100%
adpcm_c	100%	100%	100%	100%	100%
adpcm_d	100%	100%	100%	100%	100%
CRC32	√	√	√	√	√
gsm	95%	95%	95%	95%	95%

TABLE VI
NEGATIVE RUNS ON SPARC 32-BIT LINUX (GCC)

	PowerPC 32-bit Linux (GCC)				
	-O0	-O1	-O2	-O3	-Os
bitcount	√	√	√	√	√
qsort	√	√	√	√	√
susan_c	40%	40%	40%	40%	40%
susan_e	√	√	√	√	√
susan_s	75%	75%	75%	75%	75%
jpeg_c	√	√	√	√	√
jpeg_d	√	√	√	√	√
lame	√	√	√	√	√
mad	√	√	√	√	√
tiff2bw	100%	100%	100%	100%	100%
tiff2rgba	100%	100%	100%	100%	100%
tiffdither	100%	100%	100%	100%	100%
tiffmedian	100%	100%	100%	100%	100%
dijkstra	√	√	√	√	√
patricia	√	√	√	√	√
ghostscript	N/A	N/A	N/A	N/A	N/A
ispell	100%	100%	100%	100%	100%
rsynth	100%	100%	100%	100%	100%
stringsearch	√	√	√	√	√
blowfish_d	√	√	√	√	√
blowfish_e	√	√	√	√	√
pgp_d	√	√	√	√	√
pgp_e	N/A	N/A	N/A	N/A	N/A
rijndael_d	100%	100%	100%	100%	100%
rijndael_e	100%	100%	100%	100%	100%
sha	100%	100%	100%	100%	100%
adpcm_c	100%	100%	100%	100%	100%
adpcm_d	100%	100%	100%	100%	100%
CRC32	√	√	√	√	√
gsm	95%	95%	95%	95%	95%

TABLE VII
NEGATIVE RUNS ON POWERPC 32-BIT LINUX (GCC)

A. Correctness

Out of 30 benchmarks, 10 produce 100% of positive runs across all configurations: *bitcount*, *susan_e*, *jpeg_c*, *jpeg_d*, *mad*, *dijkstra*, *patricia*, *blowfish_d*, *blowfish_e*, and *pgp_d*. For all the others, there is a failure in at least one configuration on one target platform. For each of them, we try to explain the problem.

*1) **qsort**:* All configurations have positive runs, except the Windows 7 platform which shows 100% negative. This is due to the text output format and different end-of-line characters: Windows uses `CR+LF`, while Linux uses only `LF`. When using `diff -b` or `diff --strip-trailing-cr`[3], or `dos2unix` on output files, they show no difference with the absolute reference. They should be considered false negatives since the benchmark runs correctly, only the testing procedure is inappropriate. A better testing framework should either use different files for different platforms, or specify what utility is to be used for comparison, with the appropriate set of flags.

*2) **susan_c**:* This benchmark has 40% negative runs in many configurations. Fine grain analysis reveals that these outputs can be divided into two groups: a first group which includes outputs that all compare equal to the local reference (i.e. *x86 32-bit Windows cygwin+GCC* at -O0, and *x86 32-bit Linux icc* at -O0), and a second group which includes all other outputs of all other targets and configurations.

This benchmark [20] detects corners in the input image and it draws a small square around them in the output image. For 8 out of the 20 data sets, some corners are detected at slightly different locations, off by a few pixels. This leads us to suspect floating point rounding errors. We took advantage of the recently released GCC version 4.5 to experiment with the new flag `-fexcess-precision=standard`. All differences disappear, confirming our intuition.

When comparing images pixel by pixel, we notice that a very small percentage of pixels have a very different color: the number of differing pixels never exceeds 0.02%. Whether these runs should be considered false negatives is difficult to tell for the non-specialist. The paper [20] discusses this problem and reports that measuring fine localization is not appropriate. However, the algorithm was initially developed for recognizing corners and edges in magnetic resonance images of the brain. Correctness of a detection is better defined by experts in computer vision.

*3) **susan_s**:* Only the PowerPC configuration produces negative runs, but they represent 75% of the total on this configuration. The total number of differing pixels can be large, as much as 33% on data set number 6. The difference in gray level, however, never exceeds 1 (out of 256). In fact, to the naked eye, images cannot be distinguished.

Again, the difference could come from floating point rounding. Since the purpose of *susan_s* is noise reduction (smoothing), a difference of one gray level is probably acceptable, but should be confirmed by an expert. In any case, a definite test must be defined (is there any threshold to the acceptable deviation from the reference image?)

[3]Strip trailing carriage return on input.

*4) **lame**:* This benchmark produces negative runs only on the x86 Linux GCC configuration, at optimization levels -O1, -O2 and -O3.

We suspected rounding error on floating-point computations. We run this benchmark again at -O1, -O2, -O3, and -Os with the `-ffloat-store` option which forces storing floating-point variables to memory instead of registers (FP registers are 80-bit long on x86 targets): all configurations give results similar to the absolute reference for all 20 data sets. Compiling with the flag `-fexcess-precision=standard` of the new GCC 4.5 also resolves the problems, which confirms the analysis.

The benchmark *lame* is an MP3 encoder [12]. To assess the validity of the produced output, we first listened to the generated sound files. All produced files *sound* the same as the reference file. The files, however, significantly vary. On some data set, we measure that 28% of the bytes differ. Such a difference is too high a value to consider these results equivalent to their reference. Thus, an automated comparison which would tolerate some differences cannot simply rely on raw byte comparison (a 28+% threshold is not reasonable).

For a more detailed analysis, we converted both the reference and the different output to audio samples and we measured the difference between the two signals. Figure 1 illustrates the reference signal and difference between the reference and a produced output in a typical case: the signals are equal most of the time, and some glitches appear at various times (the difference is the white signal superimposing the red reference).

MP3 is a lossy compression algorithm. It is not surprising that an implementation is not bit-accurate since it is inherent to the application to drop some accuracy in favor of file size.

We quantify the difference shown on Figure 1 by using two standard norms for functions defined as follows:

$$\|f\|_1 = \int_0^\infty |f(t)|dt$$

and

$$\|f\|_\infty = \sup_t |f(t)|.$$

We measure the variations between the signals as $\frac{\|f-f_{ref}\|}{\|f_{ref}\|}$ for both norms.

Intuitively, $\|.\|_\infty$ gives a measure of the maximum variation between the signals, and $\|.\|_1$ focuses on the area between the two curves, it is more sensitive to a constant difference than to a single high-intensity difference. Table VIII reports several measures of the difference between the reference run and the ones produced at higher optimization levels (-O1, -O2 and -O3 produce the same results) on the x86 platform. The first column reports the percentage of differing bytes, the second and third columns respectively report the variations according to the two norms. $\|.\|_1$ reports lower variations, which seems more appropriate, since the audio files *sound* right. Obviously, a better metric and a threshold should be defined by the designer of the application, or an expert of this field.

*5) **tiff2bw, tiff2rgba, tiffdither, tiffmedian**:* The *tiff* benchmarks have negative runs only on SPARC and PowerPC, but on these configurations, 100% of the runs are negative.

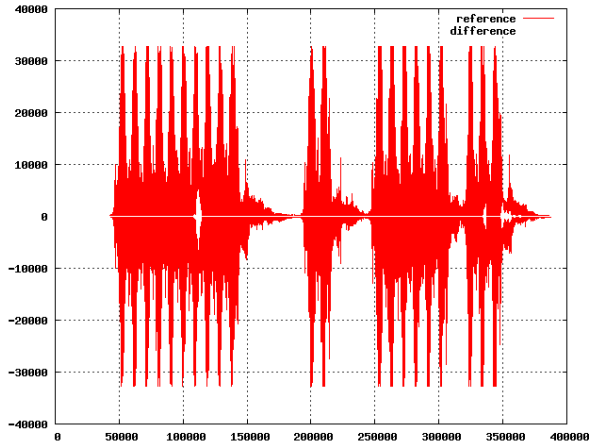

Fig. 1. Reference and difference output signals for benchmark *lame*

Data set	Bytes	$\|.\|_1$	$\|.\|_\infty$
1	1.09%	0.05%	0.74%
2	7.27%	0.30%	8.92%
3	5.99%	0.02%	0.60%
4	27.95%	0.12%	5.18%
5	13.76%	0.06%	6.29%
6	10.69%	0.05%	4.80%
7	1.70%	0.01%	0.77%
8	3.67%	0.04%	0.22%
9	0.00%	0.00%	0.00%
10	1.30%	0.00%	0.00%
11	1.77%	0.05%	15.56%
12	0.00%	0.00%	0.00%
13	11.28%	0.42%	19.46%
14	0.00%	0.00%	0.00%
15	1.36%	0.00%	0.00%
16	2.18%	0.04%	0.76%
17	1.70%	0.03%	1.98%
18	1.00%	0.00%	0.00%
19	2.94%	0.04%	1.16%
20	1.70%	0.00%	0.03%

TABLE VIII
DIVERGENCE FROM REFERENCE FOR VARIOUS METRICS, X86 LINUX (%)

It turns out that the TIFF standard [2] specifies two legal file formats, one for each endianness. For convenience, each machine stores the image file in its own endianness. This makes a straight comparison of the files across architectures impossible. Instead, we generated a local reference on each machine as the output of the benchmark, when compiled by GCC at optimization level -O0. All runs compare equal to their respective local reference, for all four *tiff* benchmarks.

This gives a false impression of correctness. When we convert our absolute reference and all outputs to the PPM format (Portable Pixmap Format), we obtain the following results: all runs of *tiff2bw*, *tiffmedian*, and *tiffdither* produce correct PPM images. Consequently, these benchmarks produce false negatives (correct output, simply encoded in a different endianness). However, all runs of *tiff2rgba* produce an incorrect PPM image, we have true negatives: visual inspection of the TIFF images reveals that they all look reddish.

*6) **ghostscript**:* This benchmark does not run on SPARC and PowerPC (Segmentation Fault). It is clearly not meant to be portable and should not be used in its current form as a benchmark. On all other configurations, only one negative is reported: data set number 8. A total of 15 pixels, out of

nearly a half million, grouped at two locations on the image. We were not able to identify the reason for this difference, but we believe that this negative could be a false negative provided an appropriate metric which should, once more, be defined by experts.

7) ***ispell****:* All runs are negative on Windows, SPARC and PowerPC.

- On SPARC and PowerPC, the output files are empty because the program exited early. The script that runs the program does not check the exit status and misses the fact that it is non-zero, indicating an abnormal end. Since the program does not report any reason for the early bail out, we investigated it by running it step-by-step in a debugger. It appeared that the spell checker dictionary could not be loaded: the dictionary used by the benchmark is in fact a pre-compiled binary file (hash) coded in little-endian, which is not compatible with the big-endian targets (*ispell* could not decode the magic number).

  Note that generating the dictionary hashes is usually part of the installation and occurs on the same machine: the build process (`make`) runs the `buildhash` utility which is part of the *ispell* distribution.

  This is definitely a true negative: the output files are incorrect because the benchmark did not run properly.
- Windows: the problem is similar to *qsort* described above. Windows uses `CR+LF` end-of-lines, Linux uses only `LF`. These are false negative, the benchmark runs correctly, only the output files are not properly compared.

It is interesting to note that a single benchmark can produce both true and false negatives.

8) ***rsynth****:* Most runs, in all configurations, produce negative results, with up to 19% differing bytes. However, the audio output files *sound* correct. The situation is very similar to the *lame* case described above. Computing the norm $\|.\|_1$ shows differences below 1.5%, suggesting that they may be acceptable. Data set 11, however, produces an audio file of a different length and is probably incorrect.

Compiling the benchmark with GCC 4.5 and the new flag `-fexcess-precision=standard`, the difference between the files decreases, with the norm $\|.\|_1$ always under 0.7%. The output of the data set 11 also has the correct length.

Again, without deeper knowledge of the application, it is difficult to decide between true and false negatives.

9) ***stringsearch****:* On Windows, 85% of outputs differ from the absolute reference. This application reads an input text and some input strings, and tries to find the strings in the input text (line by line). The output is a text file which quotes both the input strings and the input text according to the following format:

```
"<string>" is (not) in "<text>"
```

The problem is similar to *qsort* and *ispell*, but on the *input* files. We noticed that 85% of the input text files end their lines with `CR+LF` (Window-style). Since the reference outputs are produced on a Linux target where the `CR` character (aka `^M`) is not recognized as being part of the end of the line, 85% of them quote `<text>` + `^M`. Thus 85% of the reference output file lines will have the following format:

```
"<string>" is (not) in "<text>^M"
```

However, on Windows the `CR` character is part of the end of the line, so it is not quoted: 85% of the output file lines do not have the `CR` character before the final double quote (`"`). In order to decide whether we have true or false negatives, we may either remove every `^M` character from the output files, or use UNIX-style input files on UNIX platforms, and Windows-style input files on Windows platforms.

10) ***pgp_e, pgp_d****:* pgp_e fails on SPARC and PowerPC because the secret key cannot be read from the secret key file (`./secring.pgp`) which contains binary data. This is probably an endianness problem, but our knowledge of PGP is not sufficient to tell for sure.

Note that *pgp_d* emits warnings on these targets saying that it cannot find the public key in file `./pubring.pgp` to check signature integrity. Only the decryption part of the program runs well, whereas the checking of the signature fails. This means that only checking the decrypted output file is not enough to decide whether this benchmark succeeded, and we might have false positives.

11) ***rijndael_d****:* x86 64-bit, SPARC and PowerPC have 100% negative runs, but all outputs are similar to a local reference (i.e. compiled on the same machine at optimization level -O0). In addition, the results on SPARC at -O0 are similar to those at -O0 on PowerPC.

Suspecting an endianness problem, we started looking at the source code. The explanation comes from `aes.h`:

```
3. BYTE ORDER WITHIN 32 BIT WORDS
The fundamental data processing units in Rijndael
are 8-bit bytes. [...] However, Rijndael can be
implemented more efficiently using 32-bit words
to process 4 bytes at a time provided that the
order of bytes within words is known. This order
is called big-endian if the lowest numbered bytes
in words have the highest numeric significance
and little-endian if the opposite applies. This
code can work in either order irrespective of
the native order of the machine on which it
runs. The byte order used internally is set by
defining INTERNAL_BYTE_ORDER whereas the order
for all inputs and outputs is specified by
defining EXTERNAL_BYTE_ORDER, the only purpose
of the latter being to determine if a byte order
change is needed immediately after input and
immediately before output to account for the use
of a different internal byte order. In almost all
situations both of these defines will be set to
the native order of the processor on which the
code is to run but other settings may sometimes be
useful in special circumstances.
```

and further up in the same file:

```
#define INTERNAL_BYTE_ORDER AES_LITTLE_ENDIAN
#define EXTERNAL_BYTE_ORDER AES_LITTLE_ENDIAN
```

which means that this application should not be used in its current state on a big-endian target machine.

In order to get to this explanation, we had to read the source code, and understand how the application works which may require specific skills (thankfully here, a detailed comment helped us).

On x86 64-bit, benchmark outputs seem correct for most of the characters but some are non ASCII characters: these differences are not acceptable. Thus, this program is not made to run as-is on a 64-bit target machine. On the other hand, if

we compile it with `-m32`, the results all equal the absolute reference.

12) ***rijndael_e****:* This benchmark produces 100% negatives on x86 32-bit Windows, SPARC and PowerPC, but all outputs are similar to a local reference.

On SPARC and PowerPC, we easily suspect an endianness problem, based on our findings on *rijndael_e*, but not on x86 Windows.

On x86 Windows, we are able to run *rijndael_d* (the version compiled on this same target at -O0) on the output of *rijndael_e* (the version compiled on this same target at -O0), and we obtain the original files. This makes us believe that, despite the differences, the output of *rijndael_e* on x86 Windows may be valid. However, verifying it and explaining the differences requires more advanced cryptographic knowledge.

13) ***sha****:* All runs are negative on x86 64-bit, SPARC and PowerPC. On SPARC and PowerPC, outputs are similar to the local reference. A look into the source code shows that endianness is explicitly taken into account by the computation:

```
#ifdef LITTLE_ENDIAN
  <something>
#endif /* LITTLE_ENDIAN */
```

Since this benchmark is compiled exactly the same way on all targets, it is not surprising that outputs differ on big-endian target machines [4].

On x86 64-bit, each configuration of this program produces outputs which are totally different from the absolute reference, from the local reference and from each other. A look at the source code indicates that computations occur on `unsigned longs`. This type is 32-bit long on a 32-bit processor, but 64-bit long on a 64-bit processor with GCC. On the 64-bit architecture, valgrind [19] reports many uses of uninitialized values, while a 32-bit run reports no error. At last, compiling this benchmark with `-m32` on x86_64 gives good results, but this is only a workaround. We believe that this benchmark is a non-portable 32-bit application. Errors are true positives.

14) ***adpcm_c, adpcm_d****:* Both *adpcm* benchmarks fail on SPARC and PowerPC, but all compare equal to their local references (GCC at -O0 on the same machine) and SPARC results compare equal to PowerPC.

Input files of *adpcm_c* are, as reported by the `file` command: *RIFF (little-endian) data, WAVE audio, Microsoft PCM, 16 bit*. The application, however, populates an array of 16-bit values from a call to the POSIX function `read`. This is valid only on little-endian machines.

Examining the output of *adpcm_d* on big-endian targets shows files with bytes inverted in 2-byte chunks, w.r.t. reference files. This clearly demonstrates an endianness problem.

In this particular case, considering a local reference could be sensible. The signal that the benchmark converts to the ADPCM format is very different from the real signal, but the benchmark can be proved deterministic. The point of benchmarking the compression of an unrealistic signal, though, is debatable.

[4]Incidentally, since the macro `LITTLE_ENDIAN` is not defined, the correct results are the ones produced on SPARC and PowerPC; on x86 all the computations are incorrect, including the absolute reference. The fact that even our experimental setup is broken confirms that using applications without knowledge is risky.

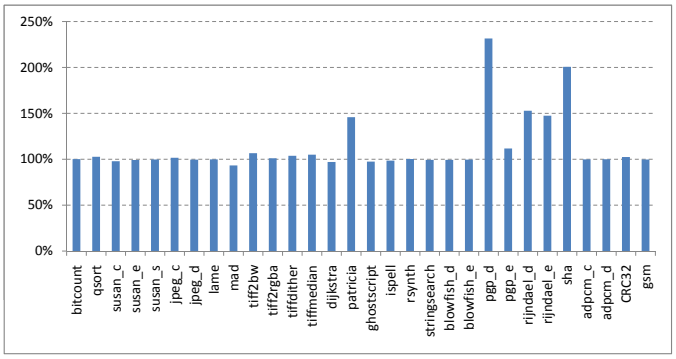

Fig. 2. Variation of CPU run time on network filesystem

15) ***CRC32****:* 100% of the output files produced on x86 64-bit differ from the absolute reference.

When looking at the output files, we realize that the CRC code computed on x86_64 is a 64-bit value with its 32 most significant bits set to 1 (`0xFFFFFFFF`), the 32 least significant bits being exactly the same as the absolute reference CRC.

Considering the source code reveals that all computations are done on `unsigned long` values which are 32-bit long on 32-bit machines, but 64-bit long on this target. The final result is printed with:

```
printf("%08lX %7ld %s\n", crc, charcnt, *argv);
```

where `crc` is an `unsigned long`.

16) ***gsm****:* 95% of the data sets produce a negative run on SPARC and PowerPC. The problem is similar to *adpcm*: the benchmark reads 16-bit samples using the `fread` function from the C library. Big-endian machines end up with the values inverted in memory.

B. Performance

After correctness, benchmarks are used to measure performance. We discuss how the structure of some applications can be misleading when it comes to measuring the run times, and we show that some understanding of the application is needed. We then discuss time distortion issues in which the relative weight of a part of an application depends on the architecture.

1) Stability of performance: In this experiment, we simply compile all the benchmarks with the GCC compiler at -O2 optimization level. The benchmarks are then run twice, on a Pentium 4 Linux workstation, clocked at 3.6 GHz, with 2 GB of memory. In the first run, all the files are on a local filesystem. In the second run, they are on a network filesystem, shared through NFS from a NetApp FAS3050 network appliance. Run times are measured with the Linux `time` utility. Table IX reports, for both configurations, the wall clock time (*elapsed time*) and the CPU time (*user+system time*) used by the benchmark. The last columns compute the ratios of elapsed and CPU times.

When run on a local file system, all benchmarks are CPU bound: the difference between elapsed and CPU times is less than 1%, i.e. within the measurement error. On a network file system, however, the benchmarks reveal different behaviors. Some are compute bound, they still run at the maximum speed

	local fs		network fs		ratios	
benchmark	elapsed	u+s	elapsed	u+s	elapsed	u+s
bitcount	9.38	9.37	9.41	9.39	1.0	1.0
qsort	11.96	11.95	12.41	12.28	1.0	1.0
susan_c	10.89	10.88	16.03	10.64	1.5	1.0
susan_e	10.75	10.74	13.58	10.65	1.3	1.0
susan_s	10.19	10.18	10.55	10.16	1.0	1.0
jpeg_c	12.07	12.06	17.73	12.24	1.5	1.0
jpeg_d	14.34	14.39	52.8	14.28	3.7	1.0
lame	14.12	14.10	14.83	14.09	1.1	1.0
mad	12.66	12.21	17.08	11.39	1.3	0.9
tiff2bw	12.31	12.29	40.23	13.10	3.3	1.1
tiff2rgba	13.92	13.91	49.23	14.06	3.5	1.0
tiffdither	12.50	12.49	28.80	12.96	2.3	1.0
tiffmedian	9.99	9.96	25.14	10.46	2.5	1.0
dijkstra	1.57	1.57	1.53	1.52	1.0	1.0
patricia	5.85	5.84	25.42	8.52	4.3	1.5
ghostscript	10.88	10.73	15.05	10.45	1.4	1.0
ispell	10.83	10.70	19.66	10.54	1.8	1.0
rsynth	13.61	13.61	13.71	13.65	1.0	1.0
stringsearch	8.53	8.52	8.5	8.47	1.0	1.0
blowfish_d	14.95	14.93	14.86	14.84	1.0	1.0
blowfish_e	14.98	14.97	14.94	14.91	1.0	1.0
pgp_d	20.79	20.77	616.99	48.08	29.7	2.3
pgp_e	12.51	12.49	368	13.97	2.9	1.1
rijndael_d	21.26	21.24	370.92	32.47	17.5	1.5
rijndael_e	22.35	22.32	416.97	32.89	18.7	1.5
sha	13.73	13.7	179.36	27.5	13.1	2.0
adpcm_c	15.25	15.25	15.26	15.23	1.0	1.0
adpcm_d	38.59	38.55	38.61	38.56	1.0	1.0
CRC32	8.55	8.54	10.45	8.74	1.2	1.0
gsm	10.52	10.51	11.04	10.49	1.0	1.0

TABLE IX
ELAPSED AND CPU (USER + SYSTEM) RUN TIMES IN SECONDS

and they do not suffer any slowdown compared to the *local* version. This is the case, for example, of *bitcount*, *dijkstra* or *blowfish*. Others incur a significant slowdown. The worst case is *pgp_d*, which runs up to 30 times slower with remote storage. The reason is that the application reads and writes small chunks of data from the file, without any buffering. The benchmark is clearly not CPU bound, and it is probably not very valuable to measure performance.

Measuring user and system CPU time instead of elapsed time helps for some of the benchmarks, but not for all of them. Recall that $user + system = elapsed \times \%CPU$. Figure 2 illustrates the variation in CPU time when executing from a network storage. In the case of *pgp_d*, we see a 2.3x slowdown of the total CPU time. A closer look reveals that most of the slowdown is due to system time (probably in the NFS stack), but the user time alone shows a 34% slowdown. Such a variation is beyond the savings most compiler optimizations or new architectural features are likely to provide.

To overcome the problem of I/Os, it is tempting to identify the computational part of a benchmark and to wrap it with a small loop. This artificially increases the importance of the algorithm compared to input and output, typically the focus of many users of benchmarks.

Some benchmarks, however, maintain a status across calls to the computation kernel. This is the case, for example, of *adpcm*. A chunk of data is read and passed to the encoder. The encoder is a differential compressor which maintains the last value encountered at the end of a chunk of data, to compare it with the first value of the next chunk. One cannot simply add

```
...
while (count >= SHA_BLOCKSIZE) {
  memcpy(sha_info->data, buffer,
                              SHA_BLOCKSIZE);
#ifdef LITTLE_ENDIAN
  byte_reverse(sha_info->data,SHA_BLOCKSIZE);
#endif /* LITTLE_ENDIAN */
  sha_transform(sha_info);
  ...
```

Fig. 3. Excerpt of the *sha* application

```
#define __LITTLE_ENDIAN 1234
#define __BIG_ENDIAN 4321
#define LITTLE_ENDIAN   __LITTLE_ENDIAN
#define BIG_ENDIAN      __BIG_ENDIAN

#define __BYTE_ORDER __LITTLE_ENDIAN
```

Fig. 4. Excerpt of endian.h

a small loop around the call to the encoder to iterate many times on each chunk of data, the value of the state would be modified.

The lack of understanding of an application makes it very difficult to modify without changing its semantics. In the case of *adpcm*, the state is a simple value. In other benchmarks, for example *blowfish*, the case is an array of values. Saving a copy of an array to exaggerate to running time of a kernel might eventually prove to bias the behavior in more subtle ways.

2) Time Distortion: The *sha* application exhibits a more subtle problem. Figure 3 shows an excerpt of the code. The application kernel is designed for big endian machines. Little endian machines are handled by simply swapping the bytes before processing them.

There is an obvious risk that a casual user of the application misses the conditional compilation directive.

It is even more unfortunate that many systems define both `LITTLE_ENDIAN` and `BIG_ENDIAN`, as shown in Figure 4. This means that both directives, incorrectly used as a boolean values, will always be considered true on all machines.

Even when properly handled, this conditional compilation directive has consequences: the actual code being compiled depends on the endianness of the target. Programs for little endian machines have an extra loop. This makes raw performance comparisons (in the spirit of the SPEC benchmarks) unfair. We recompiled the application with the `-O2 -pg` compiler flags and we used the *gprof* utility to measure the overhead of this loop. We found that the time spent in *byte_reverse* is 7.9% of the total run time for the first data set.

A similar problem in *pgp_d* is discussed above: in some cases, the benchmark is not able to read the keyfile to validate the decryption key. The benchmark still produces a correct decrypted file, though part of the benchmark was not run.

Even when relying on relative performance on a single machine, this kind of effects can bias experimental results, by distorting the speedups achieved on other parts of the application, or by artificially amplifying improvements obtained on this very simple loop.

C. Summary of the Problems

Validating the output of an application is not an easy task. In many cases, comparing files byte by byte yields false negatives, i.e. the benchmark runs correctly, even though there are minor variations in the output files. Conversely, it may be that part of the application does not run correctly, but produces a valid output, simply because this output does not derive from the entire computation (this is the case of *pgp_d* which fails to validate the decryption key).

In this paper, we have identified several reasons for false negatives.

- Text input and output files can vary slightly. End-of-line characters depend on the system: Windows uses `CR+LF`, Linux uses `LF` only.
  Although it did not occur during our tests, output can also depend on the current locale when printing dates, times, or floating point numbers (the decimal separator is a comma is many countries). File names are also likely to differ across systems.
- Floating point computation is another source of variation. Most architectures implement the IEEE 754 standard [11], thus reducing the risk of divergence. Intel architectures, however, internally use double extended 80-bit floating point arithmetic, and they may behave slightly differently when compared to strict single or double precision. Compiler optimizations like vectorization or those which assume associativity can impact results, and compilers provide many flags to control the semantics of floating point operations. Many such pitfalls are described in [17].
- Endianness can cause false negatives when the output file format authorizes different endiannesses. It is obviously more convenient for a developer to store values in the same endianness as they are in memory. This means that the benchmark can run correctly and still produce a different file.
- Benchmarks should not be sensitive to the width of basic types. Still, many applications implicitly assume 32-bit or 64-bit architectures. The type `long` of the C language is particularly error-prone.

One might be tempted to compute local references for each machine on which the benchmark is to run. However, this might hide true negatives, as happens in the *tiff2rgba* benchmark.

Suspecting floating point rounding to declare false negatives is also tempting. It may however be difficult to decide whether a program executes floating point instructions. Even though the source code contains `float` or `double` keywords, they might well be protected by conditional compilation directives (`#ifdef`). Some benchmarks, such as *jpeg*, also embed several implementations of the same routine, one of which based on floating point. The choice of the routine is made at run time, for example based on a command-line parameter.

Finally, *users* of benchmarks — compiler developers, architects — do not want, and do not have the time, to read the source code, or to run a debugger to understand if negative runs are true negatives (meaning that they introduced an error in the compiler or architecture), or if they can be classified as false negatives and ignored. Each benchmark should embed their own correctness test, including the testing framework, the metrics and the thresholds, defined by experts of the application domain.

IV. Related Work

Benchmarking is a very wide topic, and we do mean to be exhaustive. In this section, we review several benchmark suites, and also present works related to reproducibility, analysis of stability and alternatives to using real-world applications.

Many suites of benchmarks are available, representative of various kinds of workloads. Some are commercial, others are free. MiBench [8] is the suite on which we based our analysis. It has been proposed as a *set of commercially representative embedded programs* [8]. Fursin et al. [6] extend MiBench with 20 data sets for each benchmark, in an attempt to select representative samples of all possible data inputs. Refer to Section II for more details on MiBench and MiDataSets.

Unit tests for compiler developers include both commercial offers, for example ACE's SuperTest [1] and freely available tools, such as the GCC testsuite. A wide range of freely available benchmarks is available. UTDSP [15] evaluates the quality of code generated by a compiler targeting DSP processors. Mediabench [14] is a set of multimedia applications. DaCapo [4] is intended for benchmarking the Java language. PARSEC [3] is a benchmark suite for Chip-Multiprocessors (CMPs) that focuses on emerging applications. It includes a diverse set of workloads from different domains such as interactive animation or systems applications that mimic large-scale commercial workloads.

Commercial benchmark suites include EEMBC [9] (Embedded Microprocessor Benchmark Consortium) and SPEC [10]. EEMBC standardizes real-world embedded benchmarks, defined as algorithms and applications, from a variety of domains, such as telecommunication, networking, and automotive. Commercial products come for a fee, but they have been carefully adapted to be as portable as possible, and to avoid common pitfalls. SPEC, for example, has many input and output files in text form. The case of binary files is handled in two different ways. The source code of *473.astar* contains `ifdef`s that specialize I/Os to the endianness of the machine. *481.wrf* and *482.sphinx3* have two sets of input files, one for each endianness. Potential portability issues are reported, as in [10]: *403.gcc* has portability issues related to floating point, *425.gromacs* specifies that output should not differ by more than 1.25% from reference values, *999.specrand* is not expected to run on machines where `int` is 64 bits wide. This paper looks at the pitfalls of using non commercial benchmark suites.

Joshi et al. [13] propose to synthesize benchmarks that capture the properties of proprietary workloads without compromising the intellectual property. These miniature benchmarks are designed to inspire the same confidence as the real application, but are much easier to handle and to maintain.

Mytkowicz et al. [18] study the measurement bias introduced by seemingly innocuous aspects of an experimental setup. They show that the size of the UNIX environment or the linking order of object files can impact the results by a

significant amount. Their focus is on external factors, while we are interested in the benchmarks themselves.

Touati et al. [21] describe a statistical methodology to improve the reproducibility of experimental results. They introduce a statistically rigorous performance analysis and they propose a protocol to certify observed speedups. Georges et al. [7] also question the statistical rigor of current evaluation of Java applications and advocate the computation of confidence intervals. These works deal with measurement error and statistical soundness, while we are interested in distortions caused by the benchmarks themselves.

In the particular case of floating point computations, some aspects can be addressed statically. Abstract interpretation can detect potential exceptions, overflows, or invalid instructions [16]. Interval arithmetic is also able to detect stability problems in floating point computations [5], [17]. These analyses are sophisticated and time-consuming techniques. They could be used to design or validate commercial suite. But they are beyond the *user* of benchmarks.

V. Conclusion

It is not our intention to criticize MiBench, which is a popular and freely available benchmark suite used by many researchers. Rather, we use it to illustrate a number of adverse situations that can impact the large spectrum of the computing industry which relies on applications as benchmarks without any deep understanding of their behavior or implementations. This includes, in particular, compiler developers, processor architects and system integrators.

We showed that the lack of understanding can lead to invalid conclusions about the correctness or the performance of compiler optimizations. We identified several reasons for the divergence of results.

It is key to validate the outcome of benchmarks. However, in many cases, correctness cannot be reduced to a bit-wise comparison of the produced files. Correctness must be defined by the developers of the applications, based on their intimate knowledge of their application domain. A picture can be correct even if all pixels differ from the reference because their colors vary by a very small amount. Conversely, a sound file differing by one byte can be incorrect if this byte happens to encode the sampling rate.

Beyond correctness, subtle performance distortions can result from architectural features, such as endianness. They can remain hidden when large and hardly understood applications are used as benchmarks.

MiBench, and many other non-commercial benchmark suites, are still very useful. In many cases, their use is legitimate and the drawn conclusions are valid. The goal of this paper is to raise the awareness about the pitfalls of using applications as benchmarks, and to encourage the community to thoroughly check that a given benchmark suite fits their needs. We also advocate the fact that correctness should be defined by the experts of each application domain, and that testing should be integrated in the benchmarks.

References

[1] ACE Associated Compiler Experts. SuperTest C/C++ compiler test and validation suite. http://www.ace.nl/compiler/supertest.html.

[2] Abode Developers Association. *TIFF*, June 1992.

[3] Christian Bienia and Kai Li. PARSEC 2.0: A new benchmark suite for chip-multiprocessors. In *Fifth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS 2009)*, Austin, TX, USA, June 2009.

[4] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, October 2006. ACM Press.

[5] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 196–207, New York, NY, USA, 2003. ACM.

[6] Grigori Fursin, John Cavazos, Michael O'Boyle, and Olivier Temam. MiDataSets: Creating the conditions for a more realistic evaluation of iterative optimization. In *Proceedings of the International Conference on High Performance Embedded Architectures & Compilers (HiPEAC 2007)*, pages 245–260, Ghent, Belgium, January 2007.

[7] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous Java performance evaluation. *SIGPLAN Not.*, 42(10):57–76, 2007.

[8] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization*, Austin, TX, USA, December 2001.

[9] Tom R. Halfill. EEMBC releases first benchmarks. *Microprocessor Report*, May 2000.

[10] John L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, 2006.

[11] IEEE standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–58, 29 2008.

[12] International Organization for Standardization and International Electrotechnical Commission. *International Standard ISO/IEC 11172-3, 13818-3 – Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s – Part 3: Audio*, 1991.

[13] Ajay Joshi, Lieven Eeckhout, Robert H. Bell, Jr., and Lizy K. John. Distilling the essence of proprietary workloads into miniature benchmarks. *ACM Trans. Archit. Code Optim.*, 5(2):1–33, 2008.

[14] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communications systems. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 330–335, Washington, DC, USA, 1997. IEEE Computer Society.

[15] Corinna Lee. UTDSP benchmark suite. http://www.eecg.toronto.edu/∼corinna, 1998.

[16] Antoine Miné. Relational abstract domains for the detection of floating-point run-time errors. In *ESOP'04, volume 2986 of LNCS*, pages 3–17. Springer, 2004.

[17] David Monniaux. The pitfalls of verifying floating-point computations. *ACM Trans. Program. Lang. Syst.*, 30(3):1–41, 2008.

[18] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! In *ASPLOS '09: Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 265–276, New York, NY, USA, 2009. ACM.

[19] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science*, 89(2):44–66, October 2003.

[20] Stephen M. Smith and J. Michael Brady. SUSAN - a new approach to low level image processing. *Int. Journal of Computer Vision*, 23(1):45–78, May 1997.

[21] Sid-Ahmed-Ali Touati, Julien Worms, and Sébastien Briais. The Speedup Test. Technical report, Université de Versailles Saint-Quentin en Yvelines, 2010.